

Model-Based Reconfiguration Planning for a Distributed On-board Computer

Andrii Kovalov

German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
Andrii.Kovalov@dlr.de

Tobias Franz

German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
Tobias.Franz@dlr.de

Hannes Watolla

German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
Hannes.Watolla@dlr.de

Vishav Vishav

German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
Vishav.Vishav@dlr.de

Andreas Gerndt

German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
University of Bremen
Bremen, Germany
Andreas.Gerndt@dlr.de

Daniel Lüdtkke

German Aerospace Center (DLR)
Institute for Software Technology
Braunschweig, Germany
Daniel.Luedtke@dlr.de

ABSTRACT

The ScOSA project (Scalable On-board Computing for Space Avionics) of the German Aerospace Center aims at combining radiation hardened space hardware together with unreliable, but high performance COTS (commercial off-the-shelf) components as the processing nodes in a heterogeneous on-board network in order to provide future space missions with the necessary processing capabilities. However, such a system needs to cope with node failures. Our approach is to use a static reconfiguration graph that controls how software tasks are mapped to the processing nodes, and how this mapping should change in response to possible node failures.

In this paper we present a model-based approach and a tool for automatic generation of reconfiguration graphs. Based on the software and hardware models, we traverse the graph of all possible failure situations. For every node of this graph we solve a combinatorial optimization problem of mapping tasks to processing nodes either with an SMT solver or using a genetic algorithm. The resulting reconfiguration graph can then be translated into the configuration files that are deployed on the target system, eliminating the need for tedious and error-prone manual configuration design.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering**; *System modeling languages*; • **Mathematics of computing** → *Combinatorial optimization*.

KEYWORDS

Modeling, MBSE, Task-node mapping

ACM Reference Format:

Andrii Kovalov, Tobias Franz, Hannes Watolla, Vishav Vishav, Andreas Gerndt, and Daniel Lüdtkke. 2020. Model-Based Reconfiguration Planning for a Distributed On-board Computer. In *12th System Analysis and Modelling Conference (SAM '20)*, October 19–20, 2020, Virtual Event, Canada. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3419804.3420266>

1 INTRODUCTION

As technology advances, software requirements for embedded systems are rising. If a system is expected to perform many concurrent tasks, a single computing unit might not suffice. Moreover, having one CPU for all tasks can raise reliability concerns, as this introduces the single point of failure. Therefore, designing systems to be both reliable and high performance poses a challenge. For instance, in the aerospace domain, while the space-qualified hardware offers built-in fault tolerance mechanisms, its rate of execution of tasks or Instructions per Second (IPS) is subpar in comparison to commercial off-the-shelf (COTS) components [10, 18]. This challenge can be overcome by using distributed systems.

The motivation for this paper comes from the ScOSA project (Scalable On-board Computing for Space Avionics) [20]. It is a distributed approach where both reliable and high performance computing nodes form a partially connected mesh network. Software applications in this system are based on the Tasking Framework [7], and are structured as tasks that exchange messages. Each computing node can run zero or more tasks depending on the mission requirements and the current system state.

The system starts in the ‘initial’ configuration with a certain distribution of application tasks to the processing nodes. Then the system is constantly monitored for node failures. If a node fails, the system moves into the next configuration, and the tasks are remapped to the nodes that are still functional. Here and in the remainder of the paper by a ‘configuration’ we mean the assignment of tasks to the processing nodes as described in Subsection 3.3.

All configurations need to be planned in advance to ensure deterministic and reproducible behavior. These configurations form a ‘configuration set’, which is a directed decision graph with the vertices being individual configurations, and the edges being possible

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SAM '20, October 19–20, 2020, Virtual Event, Canada

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8140-6/20/10.

<https://doi.org/10.1145/3419804.3420266>

failure events. Each configuration corresponds to some subset of available nodes, so a system with n nodes can have at most $2^n - 1$ configurations (an empty set is not considered a configuration).

For a small system these configurations can be designed by hand, but as the system scales up, the number of configurations grows exponentially, quickly coming to a threshold, where it is no longer feasible to create them manually. Additionally, the decision graph itself becomes more complex, making it difficult to design and maintain. This makes the entire configuration planning cumbersome and error-prone, leading to an increase in cost and development time.

In this paper we present a model-based approach to automate configuration planning, and the tool that we are developing for this purpose.

The rest of the paper is structured as follows. In Section 2 we discuss some of the related work in fields of on-board configuration management and automatic task-node mapping. Section 3 describes how we model the system hardware, software, and deployment. In Section 4 we formulate the problem of task-node mapping and explain our approach to automatic configuration generation. Section 5 gives some insight into our modeling tool and the engineering process. In Section 6 we evaluate our approach, and Section 7 concludes the paper.

2 RELATED WORK

This section highlights some of the existing work related to certain aspects of our system.

2.1 System Modeling

An approach similar to ours can be seen in the TASTE Toolset [2] from the European Space Agency. This is a toolkit that allows modeling of data types with ASN.1, software components and hardware nodes with AADL, and can generate code in different languages for different execution platforms. However, unlike in our system, the mapping of software components to the hardware nodes has to be manually specified.

2.2 Configuration Management

In a conventional on-board computer (OBC) design with a primary and a backup processor, such as in [12], there are just two possible configurations: either the primary or the backup processor handles all the on-board tasks, so configuration management does not play a significant role.

There are more complex systems involving COTS processors, such as [14] and NASA's Dependable Multiprocessor [15, 16], where a cluster of COTS processing nodes is controlled by a reliable computer. In these systems the configurations are not precalculated, but rather every job is scheduled online to a currently available node. This approach is more flexible, but also less predictable than generating all possible configurations in advance.

An approach similar to ours is described in [5], where every processing node has a dedicated 'AMFT' (Adaptive Middleware for Fault-Tolerance) node, which monitors the state of the corresponding processing node and tracks the states of other nodes. All possible configurations are stored in the AMFT nodes, and in case of a node failure, they command processing nodes to change the task

set. As our approach, it also suffers from the exponential growth of the number of configurations.

2.3 Task-Node Mapping

Automatic mapping of tasks to processing nodes is a well-studied problem, formulations of which have been shown to be NP-complete [1, 9]. A usual approach is to use heuristic search [8], such as greedy search [3], graph partitioning [4], or genetic algorithms [21].

On smaller networks exact solvers can be successfully used, such as satisfiability modulo theories (SMT) [17, 19] or integer linear programming (ILP) solvers [22].

3 MODELING

Scientific space missions require experts from various domains working together. Scientists designing the experiments are most likely not experts in developing on-board real-time software. In our approach, models are used to help exchange information between experts from different domains. This way, hardware experts can design and model the system hardware and how it is connected. Domain experts specify which software tasks are required. Software engineers implement the tasks in software and use code generation from the model to integrate them. Finally, model-based configuration is used to deploy this software solution to different hardware setups.

This section describes our system model, which can be separated into three parts: the physical hardware, the software tasks that will run on it, and the configurations that map the tasks to the hardware.

3.1 Network

The targeted hardware is a network of nodes connected by links modeled as shown in Figure 1. Nodes can be of different user-defined types. Typical types are a reliable computing node (RCN) and a high performance node (HPN). Different operating systems can also be modeled as different node types.

Every node has a load limit between 0 and 1, which defines the percentage of the processor time the node can run application tasks. It is typically around 0.8 to allow for middleware execution, context switching, and other overhead.

Nodes are connected with bidirectional links which can be of different types (e.g. SpaceWire, Ethernet) that define the available bandwidth.

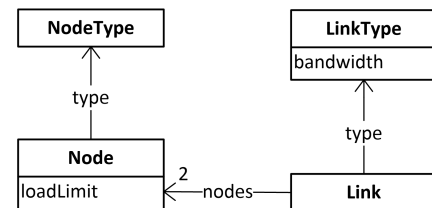


Figure 1: Hardware class diagram.

3.2 Application Tasks

Computationally demanding space missions require modularity and concurrent processing of data. Following the philosophy of the Tasking Framework, this can be achieved by creating reusable tasks that are executed in an event-driven and multithreaded way. Figure 2 shows the required components for modeling such software. All on-board software tasks are specified as task definitions that contain input and output definitions. In the on-board software, these task definitions are then instantiated and connected.

In the example in Figure 3, a system uses multiple cameras to take images at certain intervals. A camera task definition could be instantiated for each physical camera in the system. Camera tasks are then triggered by an external event and produce an image. Thus, the event is the input, and image data is the output of a camera task. Event-based execution of tasks can be customized. In this example the event has a frequency of 50 Hz. Camera1 and Camera2 directly use this frequency, but Camera3 is configured to only start with every fifth event, thus having a frequency of 10 Hz.

Task instances are connected by channels. In our example, all camera tasks could write images into a common channel, which is then read by further processing tasks. A channel is an abstraction for message exchange between tasks. Depending on their implementation, channels can act as data buffers or send messages across the network. Data transmitted by channels is specified by their message type.

Task configurations extend task instances with properties related to hardware deployment. First of all, a task configuration specifies

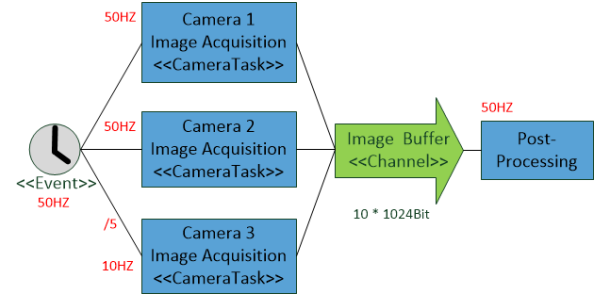


Figure 3: Example of a simple task-based architecture.

worst-case execution times (WCET) for all node types on which this task can run, as well as the period of the task.

Although the execution of task instances is event-driven (they start as soon as all required inputs are received), processing pipelines are typically triggered by periodic events, as shown in Figure 3. Thus, tasks will be also running periodically, possibly with some jitter. We need the user to specify the expected periods of all tasks because this affects how much load a task creates on its node, and how much data it produces. We use these values to assign tasks to nodes as described in Subsection 4.1.

Additionally, not all tasks can run on all nodes, and these mapping constraints are specified in properties *applicableNodes* and *applicableNodeTypes*. For example, a critical task may need to be mapped to a reliable node (constraint by node type), or a camera task should be mapped to a node that is physically connected to the camera.

Finally, some groups of tasks should be always mapped to different nodes for redundancy or parallelization purposes.

3.3 Configurations

During the course of a mission, the set of executed tasks may be different depending on operation modes and mission phases. Additionally, the on-board computer network needs to be able to adjust to node failures, while continuing to execute the required software tasks. Our goal is to prepare configurations with mapping of tasks to nodes for all possible scenarios.

Configurations are grouped into configuration sets, which normally correspond to different operation modes with different tasks. Configuration sets are independent of each other, and switching between them is a 'planned reconfiguration'. Inside a configuration set, however, switching to a new configuration is a recovery action after a node failure. Every configuration set has an initial configuration, in which all the nodes are available. Other configurations form a decision graph defining how to respond to node failures. Leaves of this graph are 'safe mode' configurations, in which there are not enough resources to run all the required tasks. In a safe mode, the spacecraft only performs the essential functions and awaits commands from the ground.

A configuration contains a number of node configurations (one for each active node), task mappings (one for each active task), and network paths for each ordered pair of remaining nodes, as shown in Figure 4.

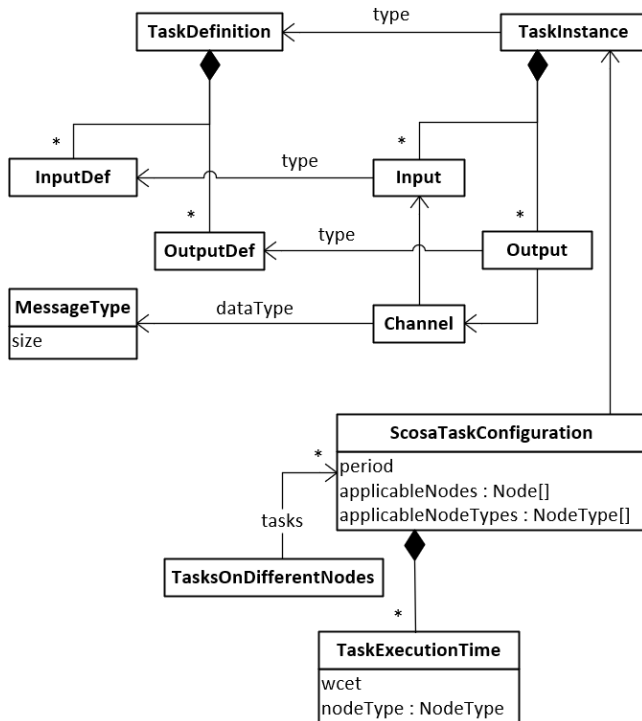


Figure 2: Class diagram for modeling software tasks.

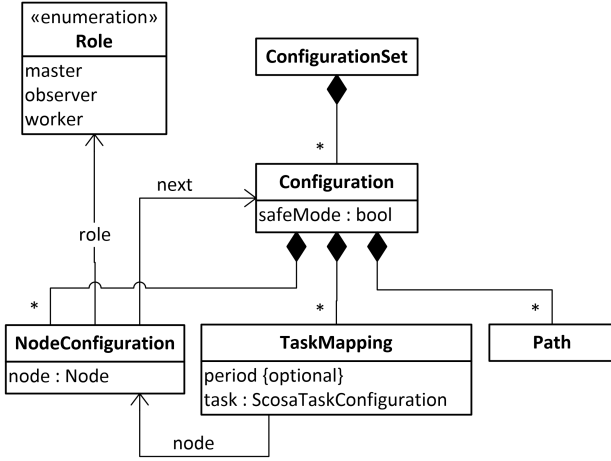


Figure 4: Configurations class diagram.

A node configuration defines a role of the corresponding node in this configuration and references the next configuration in case this node fails.

Every configuration needs one ‘master’ node that monitors the health of the other nodes and optionally one or more ‘observers’ that monitor the master. Other nodes are ‘workers’. More details on health monitoring and node roles can be found in [13].

Task mappings define how tasks are assigned to nodes and can optionally override task periods. This allows ‘graceful degradation’ when all the required tasks run with increased periods to match the available resources.

Since our network is partially connected, node failures change the network topology, so network paths need to be stored in every configuration. We model a path as a sequence of nodes from the source to the destination. Having explicit predefined paths is compatible with SpaceWire’s path addressing, the addressing scheme ScOSA is using, where the full path is included into the message header. These paths are also used to estimate the amount of network traffic.

4 RECONFIGURATION PLANNING

In this section we describe how we use the model from Section 3 to automatically generate reconfiguration graphs and individual configurations inside them.

4.1 Configuration Generation

4.1.1 Path Generation. Every configuration in a configuration set corresponds to a certain network topology of available nodes. A prerequisite for mapping tasks to nodes is the generation of network paths between all nodes. We take a simple approach of picking the shortest paths. Among all shortest paths, we select the one with links least used in the already generated paths. An example routing in a network with 4 nodes is shown in Figure 5.

4.1.2 Task-Node Mapping Problem. We formulate the problem of task-node mapping as a combinatorial optimization problem. Given

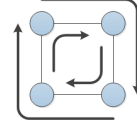


Figure 5: Routing example in a simple network. Direct paths are not shown.

a directed task graph $TG = (T, C)$, where T is a set of tasks and $C \subseteq T \times T$ is a set of directed communication channels between them, for every task $t \in T$ we define a task period $period_t \in \mathbb{R}$ and task output message size $msg_t \in \mathbb{R}$. Traffic from a producer task p to a consumer task c is then

$$tt_{pc} = \begin{cases} msg_p / period_p, & (p, c) \in C \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

Hardware network is then an undirected graph $HN = (N, E)$ consisting of a set of nodes N and a set of undirected edges E , which are communication links. Every node $n \in N$ has a load limit $ll_n \in [0, 1]$, and every link $e \in E$ has bandwidth $b_e > 0$. Worst case execution time of a task t on a node n is $wcet_{tn} > 0$, and the utilization t creates on n is defined as $u_{tn} = wcet_{tn} / period_t$.

The path from a source node s to a destination node d can be represented as a set of links a message from s to d needs to pass according to the previously picked routing, $path_{sd} \subseteq E$.

Our goal is to find a mapping function $m : T \rightarrow N$ that fulfills the following two constraints. First, the utilization on all nodes should not exceed their load limit:

$$\forall n \in N : load_n \leq ll_n, \text{ where} \quad (2)$$

$$load_n = \sum_{t \in T, m(t)=n} u_{tn}. \quad (3)$$

Secondly, the traffic on all communication links should not exceed their bandwidth:

$$\forall e \in E : traffic_e \leq b_e, \text{ where} \quad (4)$$

$$traffic_e = \sum_{\substack{s, d \in N \\ e \in path_{sd}}} \sum_{\substack{p, c \in T \\ m(p)=s \\ m(c)=d}} tt_{pc}. \quad (5)$$

Additionally, the set of available nodes for each task can be restricted with user-defined constraints.

We would like to take some objective function into account to produce an optimal mapping in some respect. In our tool we consider two possible objective functions. The first is traffic minimization trying to minimize expression $\sum_{e \in E} traffic_e$. This objective function tries to co-locate heavily communicating tasks. The second is load balancing trying to minimize $\max_{n \in N} load_n$. Of course there can be other application-driven objective functions.

This formulation does not consider many aspects of the real system such as task schedulability or peak loads. Therefore, the obtained optimal configurations need to be further verified with simulation, testing, or model checking. This is out of scope of this paper.

4.1.3 Task-Node Mapping Solvers. We developed two different solvers for the mapping problem. The first translates the problem into an SMT problem and uses an existing SMT solver Z3 to obtain the solution as explained in [11].

Since the mapping problem is NP-complete, exact solving is not feasible for large instances. For this case we implemented a genetic algorithm. First it generates an initial population of feasible solutions, which are encoded as integer arrays of length $|T|$ with an entry for each task that indicates the number of the node to which this task is mapped. Then these solutions are crossed and randomly mutated, producing offspring solutions. The best solutions (according to the objective function) are taken into the next generation. This process is repeated for a number of iterations, and in the end the best solution is selected. This approach does not guarantee the optimal solution, nor any solution at all, but can run on problem instances of any size.

4.2 Reconfiguration Tree Generation

We generate the reconfiguration tree in the following way. First, we solve the mapping problem for the initial configuration (which has to be created by the user). Then we recursively create new configurations by removing available nodes one by one in a depth-first search manner and mapping the original tasks to the remaining nodes in the created configurations. If the mapping is not possible or the network becomes disconnected, a configuration is marked as a ‘safe mode’ and not expanded further.

Figure 6 illustrates two strategies of generating reconfiguration graphs.

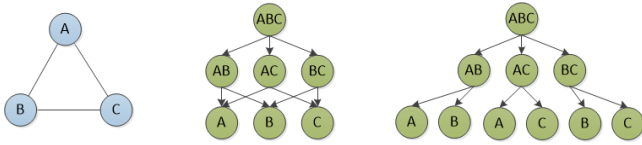


Figure 6: Example network with 3 nodes (left); ‘exponential’ reconfiguration graph (middle); ‘factorial’ reconfiguration graph (right).

In the ‘exponential’ reconfiguration graph, every configuration has a unique set of available nodes. This means a configuration can be reached via multiple paths. The number of configurations is in the order of $2^{|N|}$, and the reconfiguration graph is a subgraph of the inclusion power set graph for 2^N .

Alternatively, in a ‘factorial’ reconfiguration graph, every configuration is reachable with a unique path, and the graph itself is a tree containing in the order of $n!$ configurations.

The factorial strategy has the benefit that it can take the cost of task reallocation into account and minimize the number of tasks that need to be remapped in one reconfiguration event, potentially reducing the reconfiguration time.

On the other hand, the factorial graph grows much faster, so it cannot be used in networks larger than a few nodes. The use of this strategy can only be justified for very small networks with tasks that have high reallocation costs (e.g. long initialization).

In our tool we allow modeling of both reconfiguration graphs, however, we only support automatic generation of exponential graphs.

5 TOOL IMPLEMENTATION

One of the aims of our tool is to improve the collaboration between experts from different domains, therefore usability is highly important. The tool uses a combination of table, textual, and diagram editors to enable an intuitive and visual interaction with the model.

This section is divided into two parts. First we first describe our on-board software modeling workflow in 5.1, and then show hardware and deployment modeling in 5.2.

5.1 Software Modeling

On-board software for space systems has to fulfill high quality standards. With real-time requirements, frequency of task execution and memory management are important. Source code generated from the model is based on the Tasking Framework, which is an event-driven framework for embedded real-time applications. It provides ways to customize scheduling of tasks, event handling and concurrent execution. These parameters are integrated into the software model to provide a simplified front-end for application developers.

Figure 7 shows the modeling workflow and editors for the software part of the model. The first step is to specify atomic data types that are used in the software. Based on these atomic types, users can create data structures. The next step is to specify software tasks with their inputs and outputs. Inputs and outputs can be either pure events or messages with previously defined data structures. Such software tasks can then be instantiated in a diagram and connected with each other. Different channel types can be used to either store data or send it via the network. The editor ensures that inputs/outputs can only be connected to channels of a compatible data structure. Execution parameters of the selected task can be specified in a table editor next to the diagram.

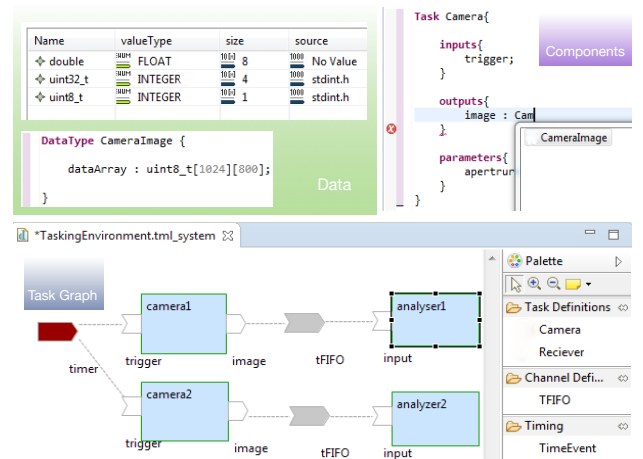


Figure 7: Model editors for the on-board software.

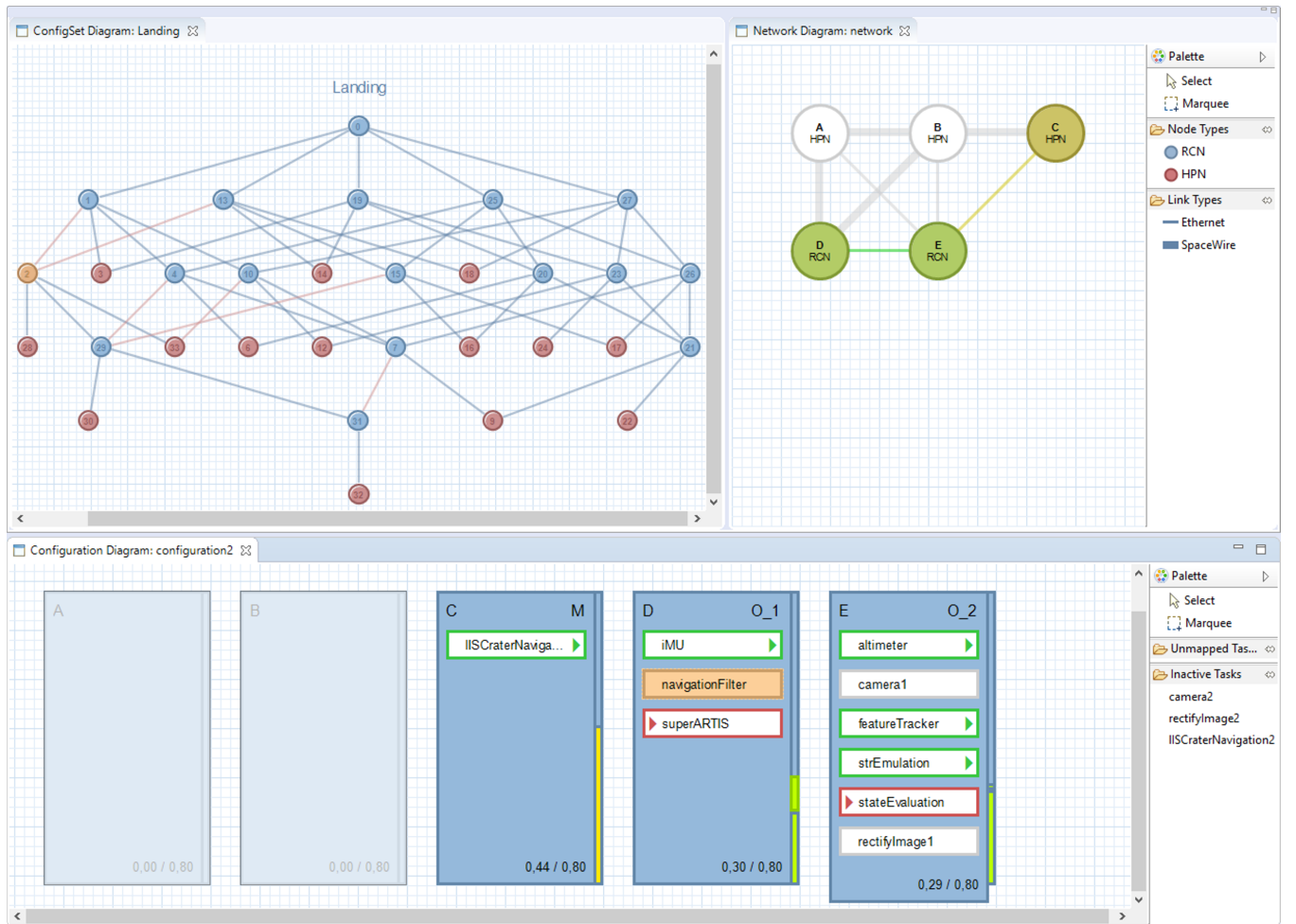


Figure 8: Deployment modeling view consisting of the Configuration Set diagram with the Reconfiguration graph (top left), the Network diagram (top right), and the Configuration diagram for mapping tasks to nodes (bottom).

The example in Figure 7 shows the specification of a *CameraImage* data type, which is used as output of a camera task definition. The camera task definition is then instantiated twice in the diagram. Both cameras are triggered by a timer and save the image data into a first in, first out channel.

5.2 Deployment Modeling

The deployment modeling is mainly done with diagram editors that are designed to be opened side by side as shown in Figure 8. These diagrams are synchronized with each other and respond to selection changes. For example, selecting a configuration in the graph brings more information about it in the network and task mapping diagrams.

The remainder of this subsection describes the modeling workflow and some usability features of these diagrams.

5.2.1 Hardware Modeling. Modeling of the computer network is done with the network editor and diagram (top right in Figure 8).

In order to start modeling the network graphically, the user first needs to create node and link types in a table editor (not shown).

When a configuration is selected in another editor, the network diagram shows the state of the network in that configuration with inactive links and nodes greyed out. Node and link colors reflect how loaded they are, similar to a heat map.

5.2.2 Configuration Modeling. The reconfiguration graph is displayed in the configuration set diagram (top left in Figure 8). Typically the user creates the initial configuration manually, and then runs a subgraph generation algorithm that creates the subsequent configurations. For every configuration the generator will try to map all tasks from the root configuration. If no such mapping can be found, the configuration is marked red as being a safe mode (application tasks are not running, and the spacecraft awaits commands from the ground).

The user can then open such a safe mode configuration and manually edit its active tasks and the task mappings in order to create a valid configuration that still meets the essential mission

requirements. For example, it is possible to deactivate certain tasks or swap a high-utilization task with a simpler version. Alternatively, it is possible to override task periods, so that the resulting task graph can be mapped to the remaining nodes. After such a manual change, it is possible to re-generate a part of the reconfiguration graph, and the manual change will be propagated to the subsequent configurations.

Links connecting configurations with different task sets are also color coded to highlight a manual change. For example, links leading to configuration 2 in Figure 8 are colored red, indicating that some tasks have been manually deactivated in this configuration. All links to the subsequent configurations are also marked red, showing that the whole subgraph has deactivated tasks.

Configurations can be viewed in detail in the configuration diagram (bottom in Figure 8). Every active node contains its respective mapped tasks. The load on nodes is visually represented with the bar on its right, and overloaded nodes are highlighted in red. The role of every node is indicated with the abbreviation in its upper right corner. The tasks can be moved around and dropped on and off the nodes to change task mapping manually. To give the user input about communication between tasks, and therefore the network traffic, tasks that are connected with the currently selected task are highlighted. In Figure 8, the task ‘navigationFilter’ is selected. All of its inputs are highlighted in green, and the tasks that wait for its output are highlighted in red.

5.2.3 Code Generation. We use our tool to generate code for the ScOSA system. This includes task stubs, configuration files, decision graphs and some other files. They are generated based on templates filled with the data from the model (see [6]).

6 EVALUATION

To evaluate our approach, we modeled and generated configurations for a system with 8 nodes and 13 tasks. The hardware nodes are arranged as shown in Figure 9, and the software model for this system is based on the ‘Autonomous Terrain-based Optical Navigation’ (ATON) project [6] and is described in more detail in [11].

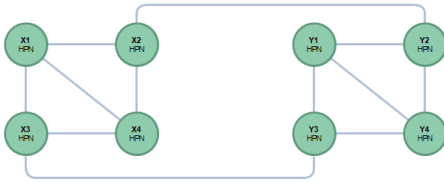


Figure 9: Network of a system with 8 high performance nodes.

The resulting reconfiguration graph is shown in Figure 10 and contains 238 configurations, out of which 115 are valid configurations and 123 are safe mode configurations. In other words, valid configurations are about 45% of all theoretically possible $2^8 - 1$ configurations. This ratio of course heavily depends on the system and how many node failures it should survive, but we can assume that the number of valid configurations also grows exponentially with the size of the network.

Modeling and configuration generation for such a system can be done by a trained user in one day. Additionally, if any changes occur at a later point, the model can be adjusted quickly, and the reconfiguration graph can be automatically regenerated.

In contrast, manual configuration planning for such a system would require significantly more effort. This would include manual task-node mapping for every configuration and arranging these configurations into a decision tree, which are tedious and potentially error-prone tasks, especially considering that a mission could need multiple reconfiguration graphs for different applications or operation modes. Therefore, we conclude that adopting our model-based approach dramatically reduces configuration planning effort in comparison to manual design.

6.1 Scalability Estimates

The limitation of our configuration management approach is the exponential number of configurations that need to be generated in advance and stored on board. In this subsection we estimate the generation time and the storage requirements for a system with n nodes, t tasks and one reconfiguration graph. For this estimate we assume that the system should be able to survive the loss of half of the nodes. Therefore, the number of configurations will be around 2^{n-1} , which corresponds to the upper half of the power set inclusion graph of n nodes.

The total generation time would then be $2^{n-1}g$, where g is the generation time of one configuration. If the task-node mapping is done with an exact solver, this g is itself exponential in the number of nodes (since the mapping problem is NP-hard). This means for large instances we would need to run a heuristic search, and we can control how long we run it (e.g. run the genetic algorithm for a certain number of generations).

As for the size, in our configuration model most information is stored in the network paths because they are defined for every pair of active nodes. However, these paths can be generated on board with the same deterministic algorithm (see 4.1.1), and can be excluded from the configuration. Then the configuration would only contain task-node mappings and node roles.

We base our estimation on the following hypothetical encoding. A role can be encoded with two bits, so we would need $2n$ bits for node roles. For every task we would need to store the number of the node it is mapped to, so the task mappings would occupy $t \lceil \log_2 n \rceil$ bits. Therefore, the total size of the reconfiguration graph would be at least $2^{n-1} (2n + t \lceil \log_2 n \rceil)$ bits, possibly with some data structure overhead. This estimate does not consider two important factors. First, some form of data compression may significantly reduce the size. Secondly, the information would need to be stored on different nodes with some degree of duplication, which would increase the total size across all nodes.

Table 1 gives some rough estimates based on these formulae for $t = 20$ and $g = 1$ second. The generation time on the ground appears to be the bottleneck of this system. Although the generation can be easily parallelized and run on a cluster, this approach does not seem feasible for networks with over 20 nodes.

One possible strategy to overcome this limitation would be to calculate and store not the whole reconfiguration graph but only the subgraph around the current system state to a certain distance.

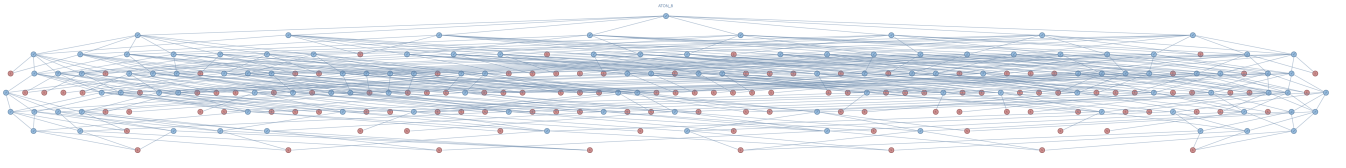


Figure 10: Generated reconfiguration graph for a system with 8 nodes and 13 tasks. It contains 238 configurations, out of which 115 are valid (blue), and the rest are ‘safe mode’ (red).

Table 1: Estimates on configuration generation time and on-board storage requirements for hypothetical systems of different size.

Nodes	Tasks	Configurations	Generation Time	Size
10	20	512	8.5 minutes	6.4 kB
15	20	16,384	4.5 hours	225 kB
20	20	half a million	6 days	9 MB
25	20	17 million	over a year	315 MB
30	20	500 million	17 years	10 GB

This would require occasional uplink of the new reconfiguration graphs from the ground.

7 CONCLUSIONS

We presented a model-based configuration planning approach for a distributed on-board computer, as well as a modeling tool that implements it. In our approach the system engineers model the hardware for the mission (a network of computing nodes) and the software tasks that should be executed. Then from these high-level models our tool generates a reconfiguration decision graph that defines how to allocate software tasks to processing nodes in response to node failure events. Based on this decision graph we can generate configuration files that can be directly deployed on the target system, therefore eliminating the need for tedious and error-prone manual configuration design.

REFERENCES

- [1] Sanjoy Baruah. 2004. Task partitioning upon heterogeneous multiprocessor platforms. In *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004. 536–543.
- [2] Eric Conquet, Maxime Perrotin, Pierre Dissaux, Thanassis Tsiodras, and Jérôme Hugues. 2010. The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software. In *European Congress on Embedded Real-Time Software (ERTS 2010)*. Toulouse, France.
- [3] Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, and Philippe O. A. Navaux. 2015. An Efficient Algorithm for Communication-Based Task Mapping. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 207–214.
- [4] Mehmet Deveci, Sivasankaran Rajamanickam, Vitus J. Leung, Kevin Pedretti, Stephen L. Olivier, David P. Bunde, Ümit V. Çatalyürek, and Karen Devine. 2014. Exploiting Geometric Partitioning in Task Mapping for Parallel Computers. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 27–36.
- [5] Muhammad Fayyaz. 2016. Task Oriented Fault-Tolerant Distributed Computing for Use on Board Spacecraft.
- [6] Tobias Franz, Daniel Lüdtkke, Olaf Maibaum, and Andreas Gerndt. 2016. Model-Based Software Engineering for an Optical Navigation System for Spacecraft. In *Deutscher Luft- und Raumfahrtkongress*. Braunschweig, Germany.
- [7] Zain Alabedine Haj Hammadeh, Tobias Franz, Olaf Maibaum, Andreas Gerndt, and Daniel Lüdtkke. 2019. Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems, In 15th annual workshop on Operating Systems Platforms for Embedded Real-Time applications. *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*, 29–34.
- [8] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. 2014. An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing. In *High Performance Computing on Complex Environments*, Emmanuel Jeannot and Julius Zilinskas (Eds.). Wiley, 75–94.
- [9] Torsten Hoefler and Marc Snir. 2011. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) (ICS '11). ACM, New York, NY, USA, 75–84.
- [10] Intel. 2020. *Intel Core 19-10900K*.
- [11] Andrii Kovalov, Elisabeth Lobe, Andreas Gerndt, and Daniel Lüdtkke. 2017. Task-Node Mapping in an Arbitrary Computer Network Using SMT Solver. In *Integrated Formal Methods*, Nadia Polikarpova and Steve Schneider (Eds.). Springer International Publishing, Cham, 177–191.
- [12] Kaspars Laizans, Indrek Sünter, Karlis Zalite, Henri Kuuste, Martin Valgur, Karl Tarbe, Viljo Allik, Georgi Olentšenko, Priit Laes, Silver Lätt, and Mart Noorma. 2014. Design of the fault tolerant command and data handling subsystem for ESTCube-1. *Proceedings of the Estonian Academy of Sciences* 63 (01 2014), 222.
- [13] Daniel Lüdtkke, Karsten Westerdorff, Kai Stohlmann, Anko Börner, Olaf Maibaum, Ting Peng, Benjamin Weps, Görschwin Fey, and Andreas Gerndt. 2014. OBC-NG: Towards a Reconfigurable On-board Computing Architecture for Spacecraft, In *Aerospace Conference*, 2014 IEEE. *IEEE Aerospace Conference*, 1–13.
- [14] Ian Vince McLoughlin and Timo Rolf Bretschneider. 2010. Reliability Through Redundant Parallelism for Micro-Satellite Computing. *ACM Trans. Embed. Comput. Syst.* 9, 3, Article 26 (March 2010).
- [15] Jeremy Ramos, John Samson, David Lupia, Ian Troxel, Rajagopal Subramaniyan, Adam Jacobs, James Greco, Grzegorz Cieslewski, John Curreri, Michael Fischer, Eric Grobelyny, Alan George, V. Aggarwal, M. Patel, and R. Some. 2006. High-performance, Dependable Multiprocessor. In *2006 IEEE Aerospace Conference*. 13 pp.–.
- [16] John Samson, Gary Gardner, David Lupia, Minesh Patel, Paul Davis, Vikas Aggarwal, Alan George, Zbigniew Kalbarczyk, and Rafi Some. 2007. High Performance Dependable Multiprocessor II. In *2007 IEEE Aerospace Conference*. 1–22.
- [17] Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. 2007. A Decomposition-based Constraint Optimization Approach for Statically Scheduling Task Graphs with Communication Delays to Multiprocessors. In *2007 Design, Automation Test in Europe Conference Exhibition*. 1–6.
- [18] BAE Systems. 2017. *RAD5545 Space VPX single board Computer*.
- [19] Pranav Tendulkar, Peter Poplavko, and Oded Maler. 2013. *Symmetry Breaking for Multi-criteria Mapping and Scheduling on Multicores*. Springer Berlin Heidelberg, Berlin, Heidelberg, 228–242.
- [20] Carl Johann Treudler, Heike Benninghoff, Kai Borchers, Bernhard Brunner, Jan Cremer, Michael Dumke, Thomas Gärtner, Kilian Johann Höflinger, Daniel Lüdtkke, Ting Peng, Eicke-Alexander Risse, Kurt Schwenk, Martin Stelzer, Moritz Ulmer, Simon Vellas, and Karsten Westerdorff. 2018. ScOSA - Scalable On-Board Computing for Space Avionics, In IAC 2018. *Proceedings of the International Astronautical Congress, IAC*.
- [21] Li Wang, Zheng Li, Miao Song, and Shangping Ren. 2012. A Genetic Algorithm based Approach to Maximizing Real-Time System Value under Resource Constraints. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*. 285–294.
- [22] Lei Yang, Weichen Liu, Weiwen Jiang, Mengquan Li, Juan Yi, and Edwin H. M. Sha. 2016. Application Mapping and Scheduling for Network-on-Chip-Based Multiprocessor System-on-Chip With Fine-Grain Communication Optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 10 (2016), 3027–3040.